Desafio 1, pergunta 1

Descrição do esquema JSON

O JSON fornecido apresenta as seguintes características:

1. Estrutura geral:
    - O JSON é hierárquico e contém objetos aninhados e arrays.
    - É composto por dois elementos principais no nível superior:
        - curUTC: Representa a data e hora no formato UTC.
        - locRef: Identifica o local associado às transações.
        - guestChecks: Um array que contém os dados detalhados dos pedidos realizados.
2. Características dos arrays:
    - guestChecks:
        - É um array de objetos, cada um representando um pedido.
        - Contém atributos como guestCheckId, chkNum, clsdFlag e totais relacionados ao pedido (subTtl, chkTtl, etc.).
        - Inclui subestruturas importantes:
            - taxes: um array de objetos, representando os impostos aplicados ao pedido.
            - detailLines: um array de objetos, contendo itens detalhados do pedido.
3. Objetos aninhados:
    - Dentro de cada detailLine, existe um objeto menuItem.
4. Tipos de dados:
    - Strings: locRef, curUTC.
    - Números: guestCheckId, chkNum, subTtl.
    - Booleanos: clsdFlag, modFlag.
    - Datas: opnBusDt, detailUTC.

Relações no esquema

- guestChecks é a entidade principal, representando os pedidos.
- taxes e detailLines são entidades associadas diretamente a cada pedido, detalhando os impostos e os itens consumidos.
- menuItem está aninhado dentro de cada detailLine, detalhando os atributos específicos de cada item.

Desafio 1, pergunta 3

Justificativa da abordagem

1. Estruturação do problema

A tarefa exigiu transformar um arquivo JSON, contendo informações de pedidos de um restaurante, em um modelo relacional eficiente. A motivação central para essa transformação baseia-se nos princípios de escalabilidade e engenharia de dados descritos nos dois livros de referência:

- Em *Foundations of Scalable Systems*, Ian Gorton explica que sistemas escaláveis devem ser projetados para crescer continuamente sem comprometer desempenho ou consistência.
- Em *Fundamentals of Data Engineering*, Joe Reis e Matt Housley destacam a importância de modelar dados de maneira que sejam úteis para consumidores finais e facilmente manipuláveis para análise e relatórios.

Esses conceitos orientaram a criação de um banco de dados relacional que atende aos requisitos de eficiência, escalabilidade e usabilidade.

2. Entendimento do JSON

O JSON analisado possui uma estrutura hierárquica, com os seguintes componentes principais:

- Raiz: inclui informações do local e timestamp.
- guestChecks: representa os pedidos realizados, contendo subtotais, totais, itens e impostos.
- detailLines: descreve os itens consumidos em cada pedido.
- taxes: detalha os impostos aplicados ao pedido.

De acordo com *Fundamentals of Data Engineering*, a normalização dessas estruturas é importante para criar um pipeline de dados eficiente e reduzir redundâncias, garantindo consistência ao longo do ciclo de vida dos dados.

3. Transformação para o modelo relacional

Com base nas características do JSON e nos princípios descritos em *Foundations of Scalable Systems*, foi desenvolvido um modelo relacional composto por quatro tabelas principais:

1. guestChecks (Pedidos): centraliza informações gerais do pedido.
2. detailLines (Itens): descreve os itens específicos consumidos.
3. menuItem (Cardápio): detalha características adicionais de cada item.
4. taxes (Impostos): registra os impostos associados ao pedido.

3.1. Justificativa para o modelo relacional

Em *Foundations of Scalable Systems*, o autor argumenta que sistemas escaláveis devem priorizar a modularidade para facilitar a manutenção e evolução futura. Esse princípio foi aplicado ao dividir os dados do JSON em tabelas específicas, minimizando redundâncias e promovendo eficiência nas consultas.

4. Abordagem técnica

4.1. Processo de extração e carga

O JSON foi carregado para o banco de dados usando uma abordagem estruturada:

1. Mapeamento de dados: as chaves do JSON foram mapeadas diretamente para colunas das tabelas relacionais.
2. Inserções incrementais: dados foram inseridos garantindo consistência referencial entre as tabelas.

De acordo com *Fundamentals of Data Engineering*, o uso de pipelines automatizados para carregar dados garante escalabilidade e reduz o risco de erros manuais durante o processo.

4.2. Normalização dos dados

O modelo foi normalizado para eliminar redundâncias e garantir integridade. Segundo Reis e Housley, a normalização é uma prática fundamental para otimizar o desempenho de consultas e reduzir o custo de armazenamento.

5. Planejamento para escalabilidade

Como sistemas de restaurantes geralmente crescem em complexidade e volume de dados, foram seguidos os princípios descritos:

- Separação de preocupações: cada aspecto do pedido (itens, impostos, totais) foi isolado em tabelas separadas, permitindo ajustes específicos sem afetar o modelo geral.
- Escalabilidade horizontal: o design suporta replicação de tabelas e distribuição de dados, caso o volume de pedidos aumente.

6. Impacto prático

A abordagem adotada oferece os seguintes benefícios:

1. Consultas otimizadas: redução no tempo de execução para relatórios e análises.
2. Manutenção simplificada: alterações futuras no JSON podem ser incorporadas sem reestruturar todo o banco.
3. Escalabilidade: o sistema está preparado para lidar com aumento de volume sem comprometer o desempenho.

7. Justificativa para a criação de tabelas adicionais

Durante o processo de transcrição do JSON para o banco de dados relacional, foi decidido criar tabelas separadas para os objetos aninhados, como discount, serviceCharge, tenderMedia e errorCode. Essa decisão foi embasada em princípios fundamentais de engenharia de dados e sistemas escaláveis, conforme discutido em *Fundamentals of Data Engineering* e *Foundations of Scalable Systems*.

Razões para a normalização

1. Facilidade de manutenção: a inclusão de objetos como discount e serviceCharge diretamente na tabela detailLines aumentaria significativamente a complexidade e a redundância dos dados. Com tabelas separadas, alterações futuras, como a adição de novos tipos de descontos ou taxas de serviço, podem ser feitas sem alterar a estrutura da tabela principal.
2. Escalabilidade, conforme apontado em *Foundations of Scalable Systems*, o crescimento do volume de dados pode afetar a performance de consultas em tabelas muito densas. Ao separar os dados em tabelas menores e relacionadas, garantimos que o sistema seja capaz de lidar com grandes volumes de transações sem degradação significativa no desempenho.
3. Integridade referencial, a criação de tabelas com chaves estrangeiras garante que todos os dados inseridos mantenham a integridade lógica. Por exemplo, um desconto (discount) ou taxa de serviço (serviceCharge) está sempre vinculado a um item detalhado válido (detailLines).
4. Flexibilidade e extensibilidade, separar as entidades permite maior flexibilidade para adicionar novos campos ou tipos de dados específicos de cada tabela sem impactar as demais. Isso segue o conceito de desacoplamento, discutido em *Fundamentals of Data Engineering*.
5. Eficiência nas consultas, consultas que envolvem apenas um subconjunto dos dados podem ser otimizadas ao operar em tabelas menores e especializadas, em vez de filtrar grandes tabelas com colunas desnecessárias.

8. Comparação com a alternativa

A alternativa de adicionar discount, serviceCharge, tenderMedia e errorCode como colunas na tabela detailLines foi considerada, mas apresenta as seguintes desvantagens:

1. Densidade da tabela: com múltiplos tipos de dados armazenados em uma única tabela, seria necessário incluir diversas colunas opcionais (nullable), levando a uma tabela densa e pouco eficiente.

2. Redundância e risco de inconsistência: a repetição de informações comuns, como códigos de desconto, seria inevitável. Isso poderia resultar em inconsistências caso os dados não fossem atualizados corretamente.
3. Limitações em análises futuras: dados aninhados em colunas tornam-se mais difíceis de serem analisados ou manipulados em sistemas de BI e relatórios.

Conclusão

A transformação do JSON para o modelo relacional seguiu os princípios de escalabilidade, modularidade e eficiência descritos em *Foundations of Scalable Systems* e *Fundamentals of Data Engineering*. Este modelo não apenas resolve o problema proposto, mas também prepara a base para integrações futuras e crescimento do sistema e da operação dos restaurantes em si.

Desafio 2, pergunta 1

A armazenagem das respostas das APIs no data lake é essencial para suportar análises avançadas, auditorias e escalabilidade da arquitetura de dados da empresa. Com base nos fundamentos descritos em *Foundations of Scalable Systems* e *Fundamentals of Data Engineering*, a justificativa inclui os seguintes pontos:

1. Preservação de histórico

Conforme descrito no *Fundamentals of Data Engineering*, armazenar os dados das APIs em um data lake garante o histórico completo das operações, que é essencial para:

- Auditorias fiscais e regulatórias.
- Investigações futuras de inconsistências nos dados.
- Comparação de períodos históricos para entender sazonalidades nas vendas.

Exemplo: a API getFiscalInvoice pode ser usada para análises tributárias retroativas, identificando padrões ou erros nos impostos arrecadados.

2. Centralização e integração

Ao centralizar as respostas de múltiplas APIs, reduz-se a complexidade de integração, facilitando:

- Análises unificadas: dados como vendas (guestChecks) e devoluções (getChargeBack) podem ser cruzados para calcular taxas reais de retenção de receita.
- Redução de latência: dados disponíveis localmente eliminam a necessidade de consultas repetidas às APIs.

Exemplo: consolidar dados de getTransactions e getCashManagementDetails permite identificar discrepâncias em pagamentos e gerenciar melhor o fluxo de caixa.

3. Escalabilidade

Conforme o *Foundations of Scalable Systems*, arquiteturas devem ser projetadas para lidar com crescimento exponencial de dados. Um data lake:

- Suporta grandes volumes de dados, reduzindo custos por usar armazenamento bruto em vez de banco de dados relacional.
- Permite integração com ferramentas analíticas, como BigQuery, para consultas escaláveis.

Cenário futuro: o aumento do número de lojas ou a expansão para outros mercados pode gerar maior volume de dados das APIs. O data lake facilita essa transição sem necessidade de reestruturação significativa.

4. Reutilização e transformação

Armazenar respostas das APIs permite transformar e reutilizar os dados para diversas análises:

- ETL otimizado: dados brutos podem ser processados em pipelines para criar tabelas analíticas no BigQuery.
- Redução de custos: evita chamadas repetidas às APIs, economizando recursos computacionais e financeiros.

Exemplo: transformar dados de guestChecks em uma tabela de métricas diárias permite calcular KPIs de vendas sem processar todo o histórico repetidamente.

5. Preparação para machine learning

O uso de um data lake facilita:

- Armazenamento de dados brutos e metadados sobre os mesmos, permitindo pré-processamento para modelos preditivos.
- Criação de pipelines automatizados para alimentar algoritmos de machine learning.

Exemplo: usar dados históricos de getTransactions para treinar modelos que preveem padrões de consumo e otimizam o menu de restaurantes.

6. Desafios e mitigações

- Custo de armazenamento: monitorar uso e excluir dados obsoletos pode otimizar custos.
- Qualidade dos dados: implementar validações e padronização ao armazenar dados das APIs.
- Segurança: garantir que o bucket do data lake esteja protegido com políticas IAM e criptografia.

Desafio 2, pergunta 2

```
restaurante-datalake/
│
├── raw/                  # Dados brutos diretamente das APIs.
│   ├── guest-checks/     # Dados relacionados aos pedidos de clientes (ERP.json aqui).
│   ├── cash-management/  # Dados relacionados à gestão de caixa.
│   ├── chargeback/       # Dados de estornos.
│   ├── fiscal-invoice/   # Dados de notas fiscais.
│   └── transactions/     # Dados de transações financeiras.
│
├── processed/            # Dados processados e preparados para análise.
│   ├── daily-revenue/    # Receita diária consolidada.
│   └── errors/           # Registros de erros encontrados no processamento.
│
├── logs/                 # Logs gerados durante o pipeline de dados.
│   ├── api-requests/     # Logs das requisições às APIs.
│   └── etl/              # Logs do processo ETL.
│
└── consolidated/         # Dados finais consolidados e prontos para consumo.
    └── ...               # Dados de alto valor, prontos para análises e relatórios.
```

Justificativa da Estrutura:

1. Divisão lógica: cada subdiretório dentro de raw/ corresponde a um endpoint ou tipo de dado fornecido pelas APIs. Isso facilita localizar e manipular os dados brutos individualmente.
2. Facilidade de manipulação: a separação em pastas permite processar cada conjunto de dados (como guest-checks ou fiscal-invoice) de forma independente, mantendo a flexibilidade para atualizações ou alterações específicas.
3. Verificação e logs: os logs em logs/api-requests/ documentam as interações com as APIs (como tempos de resposta e erros), enquanto os logs em logs/etl/ documentam possíveis falhas ou alertas no pipeline.
4. Processamento gradual: o diretório processed/ armazena dados intermediários, permitindo validações após cada etapa do pipeline (ex.: verificar receitas diárias antes da consolidação).

5. Consolidação: o diretório consolidated/ armazena dados refinados, prontos para análises ou consumo por outros sistemas. Isso pode incluir tabelas SQL ou arquivos organizados para dashboards.
6. Escalabilidade e organização: a estrutura suporta crescimento, seja por inclusão de novos endpoints ou por aumento no volume de dados, sem comprometer a organização.

Desafio 2, pergunta 3

Caso o endpoint getGuestChecks seja alterado, renomeando o campo guestChecks.taxes para guestChecks.taxation, as seguintes implicações devem ser consideradas:

1. Falhas no pipeline existente

- Impacto imediato: qualquer sistema ou pipeline que dependa do campo guestChecks.taxes falhará ao tentar processar os dados, pois esse campo não existirá mais na resposta da API.
    - Exemplos de falhas:
        - Erros em scripts Python ao acessar o campo renomeado.
        - Tabelas SQL mapeadas com o nome antigo taxes não serão mais populadas corretamente.
        - Dashboards ou relatórios que dependem desse campo apresentarão dados incompletos ou inconsistentes.

2. Atualização necessária no código

- Modificação nos scripts de ETL:
  O código que manipula a resposta da API precisará ser atualizado para refletir a nova nomenclatura do campo:
    - Substituir todas as referências a taxes por taxation.
    - Garantir que as transformações e validações realizadas no campo sejam ajustadas ao novo nome.
- Alteração no mapeamento para SQL:
  Se o campo renomeado for armazenado em tabelas SQL, os scripts de criação de tabelas e as rotinas de ingestão precisarão ser ajustados:
    - Exemplo: renomear a coluna taxes para taxation no banco de dados.

3. Necessidade de retestar o pipeline

Após a atualização do código, é necessário realizar testes completos no pipeline para garantir que:

- O campo taxation está sendo lido corretamente.

- Os dados estão sendo armazenados nos locais apropriados (tabelas ou diretórios no data lake).
- Relatórios e análises downstream estão funcionando conforme esperado.